Exceptions Are Strictly More Powerful Than Call/CC

Mark Lillibridge

July 1995
CMU–CS–95–178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as Fox Memorandum CMU-CS-FOX-95-03.

Abstract

We demonstrate that in the context of statically typed pure functional lambda calculi, exceptions are strictly more powerful than call/cc. More precisely, we prove that the simply typed lambda calculus extended with exceptions is strictly more powerful than Girard's F^ω [6, 15] (a superset of the simply typed lambda calculus) extended with call/cc and abort. This result is established by showing that the first language is Turing equivalent while the second language permits only a subset of the recursive functions to be written. We show that the simply typed lambda calculus extended with exceptions is Turing equivalent by reducing the untyped lambda calculus to it by means of a novel method for simulating recursive types using exception-returning functions. The result concerning F^ω extended with call/cc is from a previous paper of the author and Robert Harper's.

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168. The author was supported by a National Science Foundation Graduate Fellowship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: studies of programming constructs, control primitives, exceptions, recursion, λ-calculus, type theory, functional programming

1 Introduction

The relationship between the programming features of *exceptions* and *call/cc* (call with current continuation) in statically typed pure functional programming languages has been an open question for some time. Carl Gunter, *et. al*, write in a recent paper [7]:

> It is folklore (the authors know of no published proof) that neither exceptions nor continuations can be expressed as a macro in terms of the other (at least if no references are present), even though they are closely related.

In this paper we demonstrate that exceptions cannot be expressed as a macro using only call/cc in a statically typed pure functional lambda calculi, thus partially answering half of Gunter, *et. al*,'s open question. Left open is the question of whether or not exceptions can be defined using a macro in terms of call/cc and either fix, recursive types, or some similar feature.

We do this by showing that when call/cc is added to even as powerful a statically typed pure functional lambda calculi as Girard's F^ω [6, 15], the set of functions expressible in the resulting language is still a subset of the recursive functions. However, when exceptions are added to even so limited a language as the simply typed lambda calculus ($\lambda^\rightarrow$), the resulting language permits all computable functions to be expressed. (In particular, unlike in the first case, potentially non-terminating functions can be written.) This demonstrates that exceptions are strictly more powerful than call/cc for statically typed pure functional lambda calculi — not even a full global transformation on a program can reduce exceptions to call/cc.

The first of these results is a previous result of the author with Robert Harper. The second is new to this paper and involves a novel method for simulating recursive types using exception-raising functions.

2 The Power of Call/CC

In a recent paper [9], the author and Robert Harper considered an extension of F^ω with call/cc and abort, obtaining a number of results. We summarize briefly here the relevant results: Four evaluation strategies were considered, differing in whether they use call-by-name or call-by-value parameter passing and in whether or not they evaluate beneath type abstractions.

Not evaluating beneath type abstractions treats type instantiation as a significant computation step, possibility including effects. Strategies of this type are used in Quest [3] and LEAP [14], and are directly compatible with extensions that make significant uses of types at run time [11] (for example, "dynamic" types [1, 3]). Since polymorphic expressions are kept distinct from their instances, the anomalies that arise in implicitly polymorphic languages in the presence of references [16] and control operators [10] do not occur.

Strategies that evaluate beneath type abstractions are inspired by the operational semantics of ML [12]. Evaluation proceeds beneath type abstractions, leading to a once-for-all-instances evaluation of polymorphic terms. Type instantiation application is retained as a computation step, but its force is significantly attenuated by the fact that type expressions may contain free type variables, precluding primitives that inductively analyze their type arguments. The superficial efficiency improvement gained by evaluating beneath type abstractions comes at considerable cost since it is incompatible with extensions such as mutable data structures and control operators [16, 10].

All the strategies were shown to be sound except for the strategy most like ML (the call-by-value, evaluating beneath type abstractions strategy) which was shown to be unsound for full F^ω.

Restricting so that polymorphism can only be used on values, not general expressions,[1] restores soundness for this strategy. Typed CPS (continuation-passing style) transforms were then given for each strategy from the appropriate sound subset into F^ω and proven correct. Since F^ω is known to be strongly normalizing (see [6]) and the transforms are recursive functions, this implies that all programs in the original language terminate. Hence, adding call/cc to F^ω permits at most only recursive functions to be written.

It should be noted that because the simply typed lambda calculus ($\lambda^{\rightarrow}$), the polymorphic lambda calculus (F_2), and the core of ML, *Core*-ML [13, 5], are proper subsets of F^ω that this result applies to adding call/cc to them as well.

3 The Power of Exceptions

3.1 Motivation

It is standard practice when giving the semantics of untyped programming languages such as Scheme [4], to explain exceptions by use of a transform similar in spirit to a CPS transform whereby expressions returning a value of "type" τ are transformed to expressions that return a value of "type" $\tau + \sigma$ where $\alpha + \beta$ represents a sum type, the values of which are either a tag **left** and a value of type α or a tag **right** and a value of type β, and where σ is the "type" of the values carried by the exception. If the original expression evaluates to a value v then the transformed expression evaluates to the value $\mathbf{left}(\overline{v})$ where $\overline{v}$ is v transformed.[2] If, on the other hand, the original expression when evaluated raises an uncaught exception carrying the value v then the transformed expression evaluates to $\mathbf{right}(\overline{v})$.

Such a transform is easily written in the statically typed case where σ is a base type. See Figure 1, for example. Here, b is a base-type meta-variable, τ a type meta-variable, x a term-variable meta-variable, and M and N are term meta-variables. $FV(M)$ denotes the free variables of M and is used to prevent unwanted capture. Under the appropriate assumptions, it can be proved that if $M : \tau$ then $\overline{M} : [\tau]$.

However, problems arise when σ is a non-base type. If σ is an arrow type, infinite recursion results at the type level, preventing the transform from working because infinite types are not permitted in $\lambda^{\rightarrow}$. (E.g., if $\sigma = \mathbf{int} \rightarrow \mathbf{int}$ then $<\sigma> = <\mathbf{int}>\rightarrow[\mathbf{int}] = \mathbf{int}\rightarrow(<\mathbf{int}>+<\sigma>) = \mathbf{int}\rightarrow(\mathbf{int}+(\mathbf{int}\rightarrow(<\mathbf{int}> +<\sigma>))) = \ldots.$)

By adding recursive types to the destination calculus, the transform can be made to work as in Figure 2. The rules for variables, lambdas, and applications are the same as before. We use a formulation of recursive types ($\mu\alpha.\tau$) where there is an isomorphism $(\mu\alpha.f(\alpha)) \cong f(\mu\alpha.f(\alpha))$ mediated by two built in primitives:

$$\begin{array}{rcl} \mathbf{roll} & : & f(\mu\alpha.f(\alpha)) \rightarrow \mu\alpha.f(\alpha) \\ \mathbf{unroll} & : & (\mu\alpha.f(\alpha)) \rightarrow f(\mu\alpha.f(\alpha)) \end{array}$$

such that $\mathbf{unroll}(\mathbf{roll}(x)) = x$ when $x{:}f(\mu\alpha.f(\alpha))$ and $\mathbf{roll}(\mathbf{unroll}(y)) = y$ when $y{:}\mu\alpha.f(\alpha)$ where f is any function mapping types to types. For this transform, we only need one recursive type,

[1] More precisely, terms of the form $\Lambda\alpha.\, M$ are allowed only when M is a call-by-value value. (Because this strategy evaluates under lambda abstractions, $\Lambda\alpha.\, M$ is considered a value here only when M is itself a value.)

[2] More accurately, the transform evaluates to a value equal to $\mathbf{left}(\overline{v})$. A similar caveat applies to the uncaught exception case.

$$
\begin{array}{rcll}
<b> & = & b & \\
<\tau_1 \to \tau_2> & = & <\tau_1> \to [\tau_2] & \\
 & & & \\
[\tau] & = & <\tau> \; + \; <\sigma> & \\
 & & & \\
\overline{x} & = & \mathbf{left}(x) & \\
\overline{\lambda x{:}\tau.\, M} & = & \mathbf{left}(\lambda x{:}<\tau>.\, \overline{M}) & \\
\overline{M\; N} & = & \mathbf{case}\, \overline{M}\, \mathbf{of} & (x \notin FV(M)) \\
 & & \quad \mathbf{left}(x) => \mathbf{case}\, \overline{N}\, \mathbf{of} & \\
 & & \quad\quad \mathbf{left}(y) => x\; y; & \\
 & & \quad\quad \mathbf{right}(y) => \mathbf{right}(y) & \\
 & & \quad \mathbf{endcase}; & \\
 & & \quad \mathbf{right}(x) => \mathbf{right}(x) & \\
 & & \mathbf{endcase} & \\
 & & & \\
\overline{\mathbf{raise}\, M} & = & \mathbf{case}\, \overline{M}\, \mathbf{of} & \\
 & & \quad \mathbf{left}(x) => \mathbf{right}(x); & \\
 & & \quad \mathbf{right}(x) => \mathbf{right}(x) & \\
 & & \mathbf{endcase} & \\
\overline{M\; \mathbf{handle}\, x => N} & = & \mathbf{case}\, \overline{M}\, \mathbf{of} & \\
 & & \quad \mathbf{left}(x) => \mathbf{left}(x); & \\
 & & \quad \mathbf{right}(x) => \overline{N} & \\
 & & \mathbf{endcase} &
\end{array}
$$

Figure 1: Exception transform for σ a base type

$$
\begin{array}{rcl}
<b>_\alpha & = & b \\
<\tau_1 \to \tau_2>_\alpha & = & <\tau_1>_\alpha \to [\tau_2]_\alpha \\
[\tau]_\alpha & = & <\tau>_\alpha \; + \; \alpha \\
\gamma & = & \mu\alpha.\, <\sigma>_\alpha \\
 & & \\
<\tau> & = & <\tau>_\gamma \\
[\tau] & = & [\tau]_\gamma \\
 & & \\
\overline{\mathbf{raise}\, M} & = & \mathbf{case}\, \overline{M}\, \mathbf{of} \\
 & & \quad \mathbf{left}(x) => \mathbf{right}(\mathbf{roll}\, x); \\
 & & \quad \mathbf{right}(x) => \mathbf{right}(x) \\
 & & \mathbf{endcase} \\
\overline{M\; \mathbf{handle}\, x => N} & = & \mathbf{case}\, \overline{M}\, \mathbf{of} \\
 & & \quad \mathbf{left}(x) => \mathbf{left}(x); \\
 & & \quad \mathbf{right}(x) => \mathbf{let}\, x = \mathbf{unroll}\, x\, \mathbf{in}\, \overline{N}\, \mathbf{end} \\
 & & \mathbf{endcase}
\end{array}
$$

Figure 2: Exception transform using recursive types

$\mu\alpha.<\sigma>_\alpha$, so **roll**:$<\sigma>_\gamma \to \gamma$ and **unroll**:$\gamma \to <\sigma>_\gamma$ ($f = \lambda\alpha.<\sigma>_\alpha$). Aside from using this recursive type to avoid the problem of infinite types, the transform is unchanged.

It is well known that adding recursive types to the simply typed lambda calculus allows the full untyped lambda calculus to be simulated. The following encoding of the untyped lambda calculus in $\lambda^{\to}$ extended with recursive types suffices:

$$\begin{array}{rcl} \diamond & = & \mu\alpha.\,\alpha \to \alpha \\ \\ \overline{x} & = & x \\ \overline{\lambda x.\,m} & = & \mathbf{roll}(\lambda x{:}\diamond\,.\,\overline{m}) \\ \overline{m\,n} & = & (\mathbf{unroll}\,\overline{m})\,\overline{n} \end{array}$$

Here, **roll**:$(\diamond \to \diamond) \to \diamond$, **unroll**:$\diamond \to (\diamond \to \diamond)$, and under appropriate assumptions, $\overline{m} : \diamond$ for all untyped lambda calculus terms m.

A question naturally arises: since the transform suggests that exceptions carrying arrow types have an inherently recursive character, can we simulate the untyped lambda calculus using just arrow-type carrying exceptions? The following two sections answer this question in the affirmative.

3.2 Simulating recursive types with exceptions

The key idea is to use exception-raising functions to simulate the values of a recursive type. Suppose we wish to simulate values of the recursive type $\mu\alpha.\,f(\alpha)$. We will use functions of type $\star = \mathbf{unit} \to \mathbf{unit}$ and exceptions carrying values of type $f(\star)$ where **unit** is the type containing exactly one value, denoted $()$. The key definitions of **roll** and **unroll** are as follows:

$$\begin{array}{rcl} \mathbf{roll} & = & \lambda x{:}f(\star).\ \lambda y{:}\mathbf{unit}.\,(\mathbf{raise}\,x;\,()) \\ \mathbf{unroll} & = & \lambda x{:}\star\,.\,(x();\dagger)\ \mathbf{handle}\ y \Rightarrow y \end{array}$$

where $\dagger$ is any expression of type $f(\star)$. Having a term of this type is needed to make **unroll** type check. The term is never actually evaluated though unless **unroll** is called with a value not generated by **roll**, which is arguably an error. Many exception implementations (SML, for example [12]) allow their equivalent of a **raise** statement to have an arbitrary type since it will never "return". This feature can be used to construct a $\dagger$ of type $f(\star)$ if no value of type $f(\star)$ is otherwise available.

Roll packs up its argument of type $f(\star)$ and stores it in a newly created function which it then returns. This function is built so that when called it will raise an exception carrying the argument to **roll**. The extra code to return $()$ after raising the exception is solely for typing purposes as it is never executed. Since the types of the exceptions that a function may raise are not part of its type, the resulting function just has type $\mathbf{unit} \to \mathbf{unit} = \star$ as desired.

Unroll can later retrieve the value that was passed into **roll** by simply calling the new function with $()$ and catching the resulting exception which will be carrying the desired value. Hence, we have the crucial equations that $\mathbf{unroll}(\mathbf{roll}(x)) = x$ when $x{:}f(\star)$ and $\mathbf{roll}(\mathbf{unroll}(y)) = y$ when $y{:}\star$, y produced by **roll**. The later restriction on y is not a problem for the simulation because if we take a closed well-typed expression in $\lambda^{\to}$ plus recursive types and evaluate the encoding of the expression in $\lambda^{\to}$ plus exceptions, we are guaranteed that **unroll** will never be called on a term not generated by **roll**. This is because there are no values of recursive type in the original language that are not created via **roll**.

3.3 Simulating the untyped lambda calculus

By combining the well known encoding from the untyped lambda calculus to the simply typed lambda calculus plus recursive types with our method of simulating recursive types using exceptions, we obtain an encoding from the untyped lambda calculus to the simply typed lambda calculus with exceptions. In fact, it suffices for us to just have one kind of exception which carries values of type (**unit**→**unit**)→(**unit**→**unit**). Figure 3 contains (working) SML code demonstrating how to do this. The code is entirely monomorphic and uses only those features present in the simply typed lambda calculus extended with exceptions of the aforementioned type.

Since the untyped lambda calculus is Turing equivalent [2] and the encoding transform is recursive (syntax directed, in fact), this implies that the simply typed lambda calculus extended with exceptions of the above type is Turing equivalent. Hence, all computable functions can be written in it.

4 Conclusion

We have shown by a novel method that exceptions can be used to simulate recursive types. From this and the well known fact that the untyped lambda calculus can be encoded in the simply typed lambda calculus ($\lambda^{\rightarrow}$) plus recursive types, it follows that the untyped lambda calculus can be encoded in $\lambda^{\rightarrow}$ extended with exceptions. Because the untyped lambda calculus is Turing equivalent, this implies that all computable functions can be written in $\lambda^{\rightarrow}$ extended with exceptions. The ability to have exceptions of distinguishable flavors, possibly carrying values of different types, is not required.

From previous work of the author's with Robert Harper, it is known that adding call/cc to F^{ω} (a superset of $\lambda^{\rightarrow}$) preserves the fact that all programs terminate. It follows from this that only a subset of the recursive functions can be written in F^{ω} extended with call/cc. Since the set of all computable functions is proper superset of the recursive functions, the language $\lambda^{\rightarrow}$ extended with exceptions is strictly more powerful than the language F^{ω} extended with call/cc. Hence not even a full global transformation on a program can rewrite away exceptions in F^{ω} extended with call/cc.

We are grateful to Robert Harper and Mark Leone for their comments on an earlier draft of this work.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*. ACM, January 1989.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[4] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-Sep. 1991.

```
(*
 * Prepare to simulate values of the recursive type
 * \mu a. a -> a using a ML exception of type
 * (unit->unit)->(unit->unit):
 *)
type star  = unit -> unit;
type fstar = star -> star;
exception E of fstar;

fun roll(x:fstar):star =
         fn y:unit => (raise E(x); ());
fun unroll(x:star):fstar =
         (x(); (fn y:star => y)) handle E(z) => z;


(*
 * Define an encoding of the untyped lambda calculus in
 * ML using the previous simulation:
 *
 * The rules for encoding using the below functions are as
 * follows:
 *
 *    encode(x)    = x
 *    encode(\x.M) = lam(fn x => encode(M))
 *    encode(M N)  = app(encode(M),encode(N))
 *)
fun app(x:star,y:star):star  = (unroll x) y;
fun lam(x:star->star):star   = roll(x);


(*
 * As an example, we use the encoding of omega = w w
 * where w = \x.x x to write a hanging function:
 *
 * (Omega reduces to itself in one beta-reduction step,
 *  resulting in an infinite reduction sequence.)
 *)
fun hang() = let val w = lam (fn x => app(x,x))
             in app(w,w) end;
```

Figure 3: SML code to encode the untyped lambda calculus

[5] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[6] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[7] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *1995 Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, CA, June 1995.

[8] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN–CS–92–1426.

[9] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM. (Expanded journal version to appear in *Journal of Functional Programming*).

[10] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(4):361–380, November 1993. (Journal version of [8].).

[11] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[13] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[14] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

[15] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1989.

[16] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.